

Cassette tape and other magnetic tape recording without bit errors requires that we get our feet wet in the murky waters of error correcting codes . . .

How to Pick up a Dropped Bit

W Douglas Maurer
Rm 634, University Library Bldg
George Washington Univ
Washington DC 20052

The phenomenon of the dropped bit causes difficulties in two distinct areas of computer technology: in the recording of data on tape (or disk, or the like), and in the transmission of data from one place to another. Suppose, for example, that we are recording one hundred 32 bit words on a tape. Out of the 3200 bits that are to be written on the tape, there is a nonzero chance that at least one of them will be wrong. Either it will be recorded as a zero, when it should have been a one (a dropped bit) or it will be recorded as a one, when it should have been a zero (an added bit). Even if all 3200 bits are recorded correctly, there is still a nonzero chance that the next time we read this tape, we will read at least one of the one bits from the tape as if it were a zero, or one of the zero bits as if it were a one. In such a case we again speak of dropping a bit, or adding a bit. Often both dropped and added bits are referred to, generically, as dropped bits, and we shall continue to do so in this paper.

The oxide coating of a tape is sometimes unevenly distributed, particularly when the tape is old and has been used many times.

In a similar way, suppose we are transmitting a message which consists of bits. (The message does not have to involve computers at all; it may, for example, simply be a message from one Teletype to another.) In a long message there is, again, a very good chance that at least one bit which is transmitted will be received in the wrong way. It might be received as a zero, when it is supposed to be a one, or vice versa. Again we speak of bits being dropped in transmission. (One slightly confusing piece of terminology here is that the entire collection of bits, out of which a very few are dropped, is very

often referred to as a "message," even when we are not transmitting it, but rather recording it on tape or disk.)

There are many possible sources of dropped bits. Tapes often have tiny dust particles on them which interfere with the reading and writing of data. The oxide coating of a tape is sometimes unevenly distributed, particularly when the tape is old and has been used many times. The same considerations, of course, apply to floppy disk memory, or any other kind of memory involving an oxide coating. In transmitting messages from one place to another, noise in the channel and receiver can very easily degrade the quality of the reception.

In order to solve the problems created by dropped bits, we can proceed in two general classes of ways. The first is to improve our hardware in such a way that dropped bits do not occur: We can clean our tapes. We can throw away our old tapes. We can transmit messages at a slow rate, and so on. The other approach is what may be called "picking up" the dropped bits. The idea is to send a message that is longer than the original one, and that is so designed that, even if certain bits are dropped, the original message can be recovered. (As we mentioned above, the word "message" is being used here in a general sense; it may, in particular, be a record written on tape or disk.)

Picking up a dropped bit is referred to, more precisely, as "error correction," a term which must be carefully distinguished from "error detection." In error detection, we simply detect the fact that some bit has been dropped; we cannot tell which bit is the

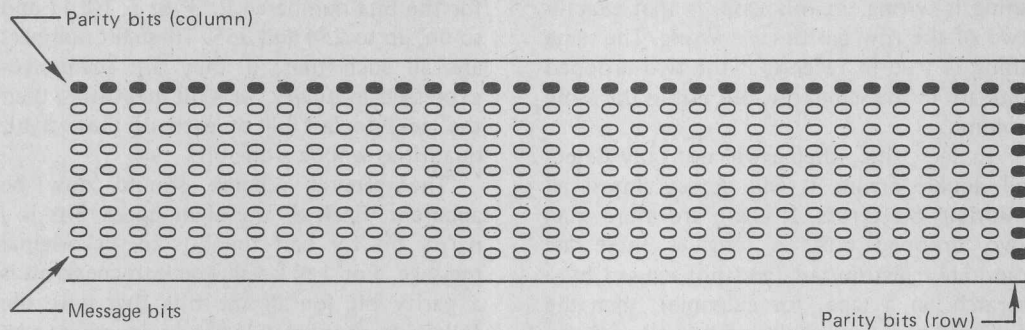


Figure 1: A block of data, eight bits plus parity in height, 32 bits plus parity in width, used as an example in the text. This layout of bytes might be thought of as a block of data on a standard 9 track tape drive; or it might be an internal memory image of data sent and received through a serial data port, bit by bit, as in the personal computer's audio tape interfaces.

wrong one. If we could tell that, then we could correct the error, because if a bit is "one," and it is wrong, then it must really be "zero," and vice versa. Knowing which bit is wrong is what enables us to do error correction.

Both error detection and error correction are affected by the question of how many bits in a message are wrong. If we can tell which bit is wrong, assuming that there is only one wrong bit, then we have *single error correction* — even though we might not be able to correct more than one error in a message (or we might correct these errors in the wrong way). In the same way, if we can tell, for example, that something is wrong whenever one or two (but no more than two) bits in a message are wrong, then we have *double error detection*, even though there might be some patterns of more than two wrong bits which are such that we cannot tell that anything is wrong.

A simple way to perform error detection is by means of what is commonly called parity checking. In large systems, 8 bit bytes stored on an IBM standard tape are always accompanied by a ninth bit, the parity bit. This bit is so chosen that the total number of one bits among the nine bits (the original eight, plus the parity bit) is an odd number (1, 3, 5, 7 or 9). (Sometimes it is done the other way around, that is, to give even parity instead of odd parity; but we shall ignore this alternative for the moment.) Now let us suppose that no more than one of these nine bits has been dropped. When the bits are read again, if the total number of one bits is an even number, we know that there has been a dropped bit. We do not know, however, which of the eight bits was dropped. (Actually, for all we know, it might be the ninth bit — the parity bit itself — that was dropped.)

Parity checking is extended, on IBM standard tapes, to provide for the possibility of error correction as well as error detection. This is done by performing "two-dimensional" parity checking, as shown in figure 1. Each column has a parity bit, and

each row also has a parity bit. Now suppose that exactly one of the bits in the message of figure 1 was dropped. If that bit was in (say) the fifteenth column and the second row, then the parity in the fifteenth column will be wrong — there will be an even number of one bits in it, rather than an odd number — and the parity in the second row will also be wrong. If these two parities are wrong, it is then a simple matter to find the bit in the fifteenth column and the second row and change it (from a zero to a one or from a one to a zero).

The assumption that is made in this parity scheme is that dropped bits will be infrequent enough that, in a message (or a record on tape) of this size, only one bit, at the most, will be dropped. For this reason the two-dimensional parity checking scheme of figure 1 is said to provide single error correction, but not double error correction. Suppose that the bit in the seventeenth column and the first row is also dropped. This means that the fifteenth column, the seventeenth column, the first row and the second row will all have wrong parity. In this case, which two bits are wrong? Let us denote by (x, y) the bit in column number x and row number y . In this case, it is bits $(15, 2)$ and $(17, 1)$ that were dropped. But it could just as easily have been bits $(15, 1)$ and $(17, 2)$, and exactly the same erroneous behavior would have occurred. In other words, we can't tell, when there is a double error, where the double error is if this scheme is used, and thus, we have no way of correcting it.

The two-dimensional parity checking scheme does, however, provide double error detection. Whenever there is a double error, that is, whenever exactly two bits in the message are dropped, we can detect this fact. This is true even when both errors are in the same column. Of course, in that case, there will be no way to tell what column the errors are in. All of the column parities will be right, because an odd number of bits with two changes in it remains an odd number of bits. The only way we can tell that some-

Both error detection and error correction are affected by the question of how many bits in a message are wrong.

Standard parity checking schemes of this kind can be improved upon in two ways: The first is by increasing their efficiency; the second is by increasing the number of errors that can be corrected.

Suppose that there are no errors at all in a given 256 bits.

thing is wrong, in this case, is that exactly two of the row parities are wrong. The same thing is true in reverse, if the two dropped bits are in the same row, but not in the same column.

In fact, this scheme will not only detect all double errors, it will detect almost all multiple bit errors. If there are more than two dropped bits, as long as these are randomly distributed (and not caused by a scratch on a tape, for example), then the probability is overwhelming that all of them, or at least most of them, will be in different columns, and thus there will be quite a number of column parities that will be wrong. We cannot refer to the given scheme as a multiple error detection scheme, in general, because there are some cases in which errors go completely undetected. For example, consider the four bits we treated earlier, namely bits (15, 2), (17, 1), (15, 1), and (17, 2). Suppose that these four bits are all dropped, and that no other bits are dropped. Now we have an undetectable error: All our parities, including those in the fifteenth and seventeenth columns and in the first and second rows, will be right. But this is so infrequent an occurrence that it may, for all practical purposes, be ignored.

Standard parity checking schemes of this kind can be improved upon in two ways: The first is by increasing their efficiency; the second is by increasing the number of errors that can be corrected. We shall treat these points one at a time.

Suppose that the record in figure 1 contained eight rows and 32 columns, for a total of 256 bits. If we include the check bits there are nine rows and 33 columns. This means that there are $9+33=42$ different check bits. (The row of check bits has to have a check bit of its own, of course, and so does the column of check bits. Sometimes these are the same, but even if they are, there are still 41 check bits.) In contrast, we will now exhibit a clever scheme that requires only eight check bits. It performs single error correction, just as does the scheme of figure 1. It has, however, certain disadvantages which we will discuss later.

The scheme is as follows. We number the bits in our message from 0 to 255. The last of the eight check bits will be a parity bit for half of the bits in the message, namely the bits numbered 1, 3, 5 and so on up to 255. Note that these are the bits such that the bit number (1, 3, 5 and so on), when it is itself expressed in binary, as an 8 bit quantity, has a one bit in the last position (indicating that it is an odd number).

The next to last of the eight check bits will again be a parity bit for half the bits in the message. This time, however, it will be

for the bits numbered 2, 3, 6, 7, 10, 11 and so on, up to 254 and 255. These bit numbers are all such that, if they are themselves expressed in binary, as 8 bit quantities, then the next to last bit of each of these 8 bit quantities will be a one bit.

The general scheme should now be apparent. Each of the eight check bits is a parity bit for half the bits in the original message. For $1 \leq k \leq 8$, the k -th check bit is a parity bit for all the bits that have the following property: If the bit number is N , and if N is expressed as an 8 bit binary quantity, then the k -th bit of this quantity is a one bit. In particular, the first of the eight check bits is a parity bit for bits 128, 129, 130 and so on, up to 255, of the original message.

Suppose now that one of our bits is dropped. For definiteness, let us suppose that it is the 99th bit. We express the number 99 as an 8 bit binary quantity: 01100011. And now let us look at our eight check bits. Which ones of them are going to be wrong? The last one will be wrong, because 99 is an odd number, and therefore the 99th bit (which was dropped) is one of the 128 bits (half of the original 256) of which the parity was taken to form this last check bit. The next to last check bit will also be wrong, because 99 has the property that the second bit from the right in its binary representation is a one bit. The first check bit, though, will still be right, because this represents parity on the 128th, 129th, 130th, etc, bits, and none of these bits were dropped.

The general pattern should now be apparent. If we look at the eight check bits from left to right, and if we write a zero for each parity check that was right, and a one for each parity check that was wrong, we obtain the pattern 01100011. This is exactly the number 99 expressed in binary. And this means that we can tell that it was, in fact, the 99th bit that was dropped — which, in turn, means that we can correct the error. Thus we have a single error correction scheme, just as before, enabling us to pick up one dropped bit.

There are three problems with this scheme. The first is as follows. Suppose that there are no errors at all in a given 256 bits. Then, of course, all the check bits will be right, and we will obtain the pattern 00000000. But this pattern tells us that bit number 0 is wrong! In fact, none of our check bits involve parity on bit number 0 at all, and thus we have no way of telling whether this bit was dropped or not. This problem, however, can be solved rather simply. We transmit only 255 bits of data, instead of 256, and these are numbered from

1 to 255. The efficiency of the scheme is not too badly affected; we now have eight check bits for every 255 data bits, rather than eight for every 256.

The second problem with the scheme is that we have not considered the possibility that one of the check bits might be wrong. Suppose that the last check bit is wrong, and that all the other bits are right. Then using the scheme above, we would obtain the 8 bit pattern 00000001, and this would tell us that it was bit number 1 of the original 256 that was wrong. In general, an error in any of the bits numbered 1, 2, 4, 8, 16, 32, 64 or 128 can be confused with an error in one of the check bits. The solution, however, is again very simple: We just leave these bits out. We are now transmitting only $256 - 8 = 247$ bits of data, with eight check bits, and the efficiency is not too badly affected. If the 8 bit pattern we obtain contains seven 0 bits, and only one 1 bit, then we know that it is a check bit that is wrong.

The third problem, however, is more serious. Suppose that more than one bit out of the 256 was dropped. Note that in our scheme, if there is an error of any kind, we determine one particular bit position to change. In other words, we always assume that if there is an error, it is a single error. If there is a multiple error we are always going to do the wrong thing. This is in contrast with the two-dimensional parity checking scheme, in which we almost always know that something is wrong, no matter how many bits get dropped. The only solution to this problem is a partial one: We can forget about correcting errors and use this scheme to detect errors only; and if we do this, all double errors will be detected. In other words, this scheme can be used for single error correction or double error detection, but not both.

A remarkable property of our scheme is that the eight check bits can all be generated simultaneously. We take the exclusive OR of all the binary integers 00000001 thru 11111111 (or 1 thru 255 in decimal) — leaving out 1, 2, 4, 8, 16, 32, 64 and 128, as noted above — which correspond to one bits in the message. That is, if the i -th bit in the message is a one bit, then the integer i , written in binary, is exclusive ORed with all other integers i with the same property. The resulting 8 bit quantity consists of precisely the eight check bits we need. For $1 \leq k \leq 8$, the k -th bit of this quantity is the exclusive OR of as many one bits as there are positions i in the message, such that bit i is a one bit and the k -th bit of the integer i is also a one bit, together with a number of zero bits, which do not affect the exclusive

OR. An algorithm for performing this process is as follows:

1. Initialize so as to point to the first bit of the message.
2. Set $R1 = 3$. ($R1$ will contain the index i as above.)
3. Set $R2 = 0$. ($R2$ will contain the eight check bits.)
4. Set $R3 = 4$. ($R3$ will be 4, 8, 16, 32, etc, as above.)
5. If the current bit in the message is a zero bit, skip the next step (that is, go to step 7).
6. Set $R2$ equal to the exclusive OR of $R2$ and $R1$.
7. Point to the next bit of the message.
8. Set $R1 = R1 + 1$.
9. If $R1 \neq R3$ then go to step 5.
10. Set $R3 = R3 + R3$.
11. If $R3 \neq 512$ then go to step 7.

At this point, if we are writing a message, we append the eight check bits in $R2$ on to the end of the message. If we are reading a message, we read the next eight bits and form the exclusive OR of these bits with $R2$. If the result is zero, the message is without error. If it is 1, 2, 4, 8, 16, 32, 64 or 128, then one of the check bits is wrong. If it is anything else — call it i — then the i -th bit is wrong, and must be changed (to a zero if it is a one, or vice versa).

It should also be clear that there is a scheme like this for any number m of check bits. We have here taken $m = 8$, and the number of data bits is $2^m - m - 1 = 256 - 8 - 1 = 247$; but we could have taken $m = 4$, for example, obtaining four check bits for each 11 bits of data. This provides another approach to the problem of multiple errors in 256 bits; we can require only that there be no multiple errors in 11 bits (say), at the cost of a certain loss of efficiency.

Is it possible to pick up more than one dropped bit at a time? That is, can we devise a scheme that is capable of double error correction? Yes, we can; we can even provide n -tuple error correction, for any (fixed) positive integer n . Schemes for doing this, however, are quite complex, and their complexity increases with the number of errors to be corrected. There is a whole subfield of electrical engineering called the theory of error correcting codes, which concerns itself with schemes of this kind. It is remarkable that error correcting codes involve one of the few known practical applications of the theory of Galois fields. (Every mathematician knows the tragic story of Galois, a French math student back in the Age of Dueling who got involved in a challenge to a duel, and, knowing his opponent was a far better duelist than he, spent his last night on earth

Suppose that more than one bit out of the 256 was dropped.

Is it possible to pick up more than one dropped bit at a time?

A fundamental concept in all error correcting code is "Hamming distance."

feverishly writing down all the mathematics he could. He died at 21, a monument to the stupidity of taking politics too seriously.)

A fundamental concept in all error correcting codes is "Hamming distance." Consider two code words C_1 and C_2 ; each code word consists of a message (data bits) together with the check bits for that message. Take the exclusive OR of C_1 and C_2 ; the number of one bits in the result is called the Hamming distance between C_1 and C_2 . If the Hamming distance is 1, this means that C_1 and C_2 are the same, except for one bit position at which they are different. This in turn means that if it was that particular bit which was dropped, then C_1 will get mistaken for C_2 , or vice versa.

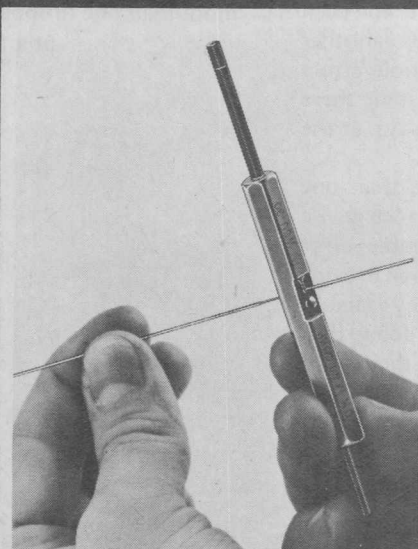
On the other hand, suppose that a particular code has the property that for every pair of code words C_1 and C_2 , the Hamming distance is 3 or more. Now suppose that C_1 is a code word and C_2 is not. Suppose that when a message is transmitted, then, due to some bit being dropped, it is C_2 that is received when it should have been C_1 . That is, the distance between C_1 and C_2 is 1 (since only one bit was dropped). In this case the error can always be corrected. That is, of all possible code words, we can always tell that C_1 is the one we wanted. To prove

this, suppose that there were another code word, C_3 , that is actually the one we wanted. Then the distance between C_3 and C_2 would be 1 (since we are assuming that only one bit is dropped), and we already know that the distance between C_1 and C_2 is 1. But in this case the distance between the two code words C_1 and C_3 cannot be greater than 2, and this contradicts our assumption that two code words must have a distance between them of 3 or more.

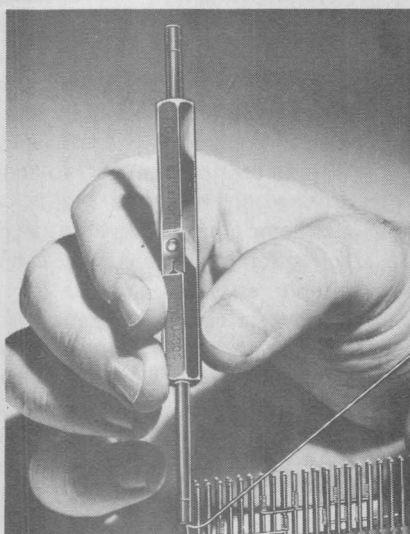
In general, if the minimum Hamming distance between any two code words is 3, the code is a single error correcting code (although the actual correcting of the errors might, in some cases, be an elaborate and inefficient process). We can extend this immediately and say that, if the minimum distance is $d = 2e+1$, the code is an e -tuple error correcting code (usually referred to as an e -error correcting code), for any integer e . This code will not detect any more than e errors, unless we sacrifice some error correction capability. If x and y are integers with $x > y$ and $x+y+1 = d$, then we can use a code with minimum distance d , as above, to correct any y errors and simultaneously detect any x errors. As a special case of this, if $y = 0$, we can detect any $d-1$ errors without any error correction capability at all. ■

IN WIRE-WRAPPING HAS THE LINE...

HOBBY-WRAP-30 WIRE-WRAPPING, STRIPPING, UNWRAPPING TOOL FOR AWG 30 (.025 SQUARE POST)



STRIP



WRAP



UNWRAP

\$5⁹⁵

SHIPPING CHARGE \$1.00
NEW YORK STATE RESIDENTS ADD SALES TAX

OK MACHINE & TOOL CORPORATION

3455 CONNER STREET, BRONX, NEW YORK, N.Y. 10475 U.S.A. • PHONE (212) 994-8600

TELEX: 125091 TELEX: 232395